Introduction

The banking sector plays a pivotal role in the global economy, facilitating financial transactions, providing various financial services, and serving as a crucial pillar for economic stability and growth. In the modern era, with the advent of technology and changing consumer preferences, the banking industry has undergone significant transformation, leading to the emergence of innovative business models and customer-centric approaches.

In this assignment, we delve into the realm of banking management by conceptualizing a comprehensive business model that encapsulates the core functionalities and operations of a banking institution. Our scenario revolves around the establishment of a fictitious banking entity aimed at catering to the diverse financial needs of individuals and businesses.

The primary objective of our banking management business model is to design and implement an efficient, secure, and user-friendly platform that enables customers to perform various banking activities seamlessly. From account creation and fund management to transaction processing and customer service, our application encompasses a wide array of features to deliver a holistic banking experience.

Scenario

In this Java application, we aim to create a Bank Management System that provides basic functionalities similar to those found in a typical bank. The system will allow users to manage checking and savings accounts, perform transactions such as deposits and withdrawals, and view transaction history.

Here's how the system will work:

Account Management:

- Users can create new checking and savings accounts.
- Each account will have a unique account number and an initial balance of zero.

Deposits and Withdrawals:

- Users can deposit funds into their accounts.
- They can also withdraw funds from their accounts, provided they have sufficient balance.

Transaction History:

- The system maintains a transaction history for each account.
- Users can view their transaction history to track all deposits, withdrawals, and other account activities.

Interest Calculation (for Savings Accounts):

- Savings accounts earn interest on their balances.
- The system automatically calculates and applies interest to savings accounts periodically.

User Interaction:

- The application provides a user-friendly interface for users to interact with their accounts.
- Users can access various functionalities through a menu-driven interface.

Design

The Bank Management System is designed using object-oriented principles, with several classes responsible for different aspects of the system. The classes involved in the system design are as follows:

Bank Class:

- Manages accounts and transactions.
- Contains an array of accounts (CheckingAccount and SavingsAccount).
- Provides methods to add new accounts, retrieve accounts, and perform account-related operations.

Account Class (Abstract):

- Serves as the parent class for specific account types.
- Contains common attributes and methods shared by all account types.
- Includes attributes such as account number, balance, and transaction history.
- Provides abstract methods for deposit and withdrawal operations, which will be implemented by child classes.

CheckingAccount Class (Subclass of Account):

- Represents a checking account.
- Inherits from the Account class and implements deposit and withdrawal methods specific to checking accounts.
- Includes a transaction fee for each transaction.

SavingsAccount Class (Subclass of Account):

- Represents a savings account.
- Inherits from the Account class and implements deposit and withdrawal methods specific to savings accounts.
- Includes an interest rate for calculating interest on balances.

Transaction Class:

- Represents a single transaction performed on an account.
- Contains attributes such as transaction ID, account number, amount, transaction type, and timestamp.
- Used to record individual transactions in the transaction history of accounts.

TransactionType Enum:

- Defines various transaction types such as deposit, withdrawal, and interest.
- Used by the Transaction class to categorize different types of transactions.

AccountDriver Class:

- Acts as the main driver class for the Bank Management System.
- Provides a user-friendly interface for users to interact with the system.
- Implements menus and user input handling for account operations.

The design ensures a clear separation of concerns, with each class responsible for specific functionalities within the Bank Management System. The inheritance hierarchy allows for code reuse and scalability, while interfaces and abstract methods ensure consistent behavior across different account types. Additionally, enums are used to define and manage transaction types, enhancing code readability and maintainability.

Code

In this section, we provide the complete implementation of our banking management system. The system is designed to handle various banking operations including account management and transaction handling. The code is organized into multiple classes, each serving a specific purpose in the overall system. Below is an outline of the classes and their functionalities:

Main Class (AccountDriver)

```
import java.util.Scanner;

import java.util.List;

public class AccountDriver {


  // Entry point of program

  public static void main (String [] args) {


    Scanner keyboard = new Scanner (System.in);


    // Create Bank object to manage accounts

    Bank bank = new Bank (100000);


    int choice;


    do {

        choice = menu(keyboard);

      System.out.println();
    switch(choice) {
        case 1:
            bank.addAccount(createAccount(keyboard));
            break;
        case 2:
            doDeposit(bank, keyboard);
            break;
        case 3:
            doWithdraw(bank, keyboard);
            break;
        case 4:
            applyInterest(bank, keyboard);
            break;
        case 5:
            printTransactionHistory(bank, keyboard);
            break;
        case 6:
            System.out.println("Thank You for using CITY BANK !!!");
            break;
        default:
            System.out.println("Invalid choice. Please try again.");
            break;
    }
    System.out.println();
} while (choice! = 6);

keyboard.close(); // Close the scanner
```

```
}
public static int accountMenu(Scanner keyboard) {
   System.out.println("Select Account Type");
   System.out.println("1. Checking Account");
   System.out.println("2. Savings Account");

   int choice;
   do {
      System.out.print("Enter choice: ");
      choice = keyboard.nextInt();
   } while (choice < 1 || choice > 2);

   return choice;
}

/**
 * Function to perform Deposit on a selected account
 */

public static void doDeposit(Bank bank, Scanner keyboard) {
   // Get the account number
   System.out.print("\nPlease enter account number: ");
   int accountNumber = keyboard.nextInt();

   Account account = bank.getAccount(accountNumber);
   if (account! = null) {
      // Amount
```

```
          System.out.print("Please enter Deposit Amount: ");

          double amount = keyboard.nextDouble();

          account.deposit(amount);

          System.out.printf("Deposit of $%.2f successfully made to account %d%n", amount,
accountNumber);

        } else {

          System.out.println("No account exist with AccountNumber: " + accountNumber);

        }

    }


    public static void doWithdraw(Bank bank, Scanner keyboard) {

        // Get the account number

        System.out.print("\nPlease enter account number: ");

        int accountNumber = keyboard.nextInt();


        Account account = bank.getAccount(accountNumber);

        if (account! = null) {

          // Amount

          System.out.print("Please enter Withdraw Amount: ");

          double amount = keyboard.nextDouble();

          account.withdraw(amount);

          System.out.printf("Withdrawal of $%.2f successfully made from account %d%n", amount,
accountNumber);

        } else {

          System.out.println("No account exist with AccountNumber: " + accountNumber);

        }

    }
    public static void applyInterest(Bank bank, Scanner keyboard) {
        // Get the account number
        System.out.print("\nPlease enter account number: ");
        int accountNumber = keyboard.nextInt();

        Account account = bank.getAccount(accountNumber);
        if (account! = null && account instanceof SavingsAccount) {
            ((SavingsAccount) account).applyInterest();
            System.out.printf("Interest applied to account %d%n", accountNumber);
        } else {
            System.out.println("No or invalid savings account exist with AccountNumber: " +
accountNumber);
 }
    }

    public static Account createAccount(Scanner keyboard) {
        Account account = null;
        int choice = accountMenu(keyboard);

        int accountNumber;
        System.out.print("Enter Account Number: ");
        accountNumber = keyboard.nextInt();

        if (choice == 1) { // checking account
            System.out.print("Enter Transaction Fee: ");
            double fee = keyboard.nextDouble();
            account = new CheckingAccount(accountNumber, fee);
            System.out.printf("Checking Account created with Account Number: %d and Transaction
Fee: $%.2f%n", accountNumber, fee);
        } else { // savings account
            System.out.print("Please enter Interest Rate: ");
            double ir = keyboard.nextDouble();
            account = new SavingsAccount(accountNumber, ir);
            System.out.printf("Savings Account created with Account Number: %d and Interest
Rate: %.2f%%%n", accountNumber, ir);
        }
        return account;
    }

    public static void printTransactionHistory(Bank bank, Scanner keyboard) {
        // Get the account number
        System.out.print("\nPlease enter account number: ");
        int accountNumber = keyboard.nextInt();

        Account account = bank.getAccount(accountNumber);
        if (account != null) {
            List<Transaction> transactions = account.getTransactions();
            System.out.println("Transaction History for Account " + accountNumber + ":");
            for (Transaction transaction : transactions) {
                System.out.println(transaction);
            }
        } else {
            System.out.println("No account exist with AccountNumber: " + accountNumber);
        }
    }
    /**
     * Menu to display options and get the user's selection
     */

    public static int menu (Scanner keyboard) {
        System.out.println("Bank Account Menu");
        System.out.println("1. Create New Account");
        System.out.println("2. Deposit Funds");
        System.out.println("3. Withdraw Funds");
        System.out.println("4. Apply Interest");
        System.out.println("5. Show Transaction History");
        System.out.println("6. Quit");

        int choice;

        do {
            System.out.print("Enter choice: ");
            choice = keyboard.nextInt();
        } while (choice < 1 || choice > 6);

        return choice;
    }
}
```

Bank Class

```
import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;


public class Bank {

   private Account [] accounts;

   private List<Transaction> [] transactionHistory;

   private int numAccounts;


   @SuppressWarnings("unchecked")

public Bank (int capacity) {

   if (capacity <= 0) {

      throw new IllegalArgumentException("Capacity must be greater than 0");

   }

   this.accounts = new Account[capacity];

   this.transactionHistory = new ArrayList[capacity];

   this.numAccounts = 0;

}


   public void addAccount(Account account) {

      if (numAccounts < accounts.length) {

         accounts[numAccounts] = account;

         transactionHistory[numAccounts] = new ArrayList<> ();

         numAccounts++;

      } else {
            System.out.println("Maximum account limit reached. Unable to add new accounts.");
        }
    }
    public Account getAccount(int accountNumber) {
        for (int i = 0; i < numAccounts; i++) {
            if (accounts[i]. getAccountNumber() == accountNumber) {
                return accounts[i];
            }
        }
        return null; // Account not found
    }

    public Account [] getAllAccounts() {
        return Arrays.copyOf(accounts, numAccounts);
    }

    public void deposit (int accountNumber, double amount) {
        Account account = getAccount(accountNumber);
        if (account! = null) {
            account.deposit(amount);
            transactionHistory[getAccountIndex(accountNumber)]. add (new Transaction
  (transactionHistory.length + 1, accountNumber, amount, TransactionType.DEPOSIT));
        } else {
            System.out.println("Account not found.");
        }
    }
```

```
}

    public void withdraw (int accountNumber, double amount) {
        Account account = getAccount(accountNumber);
        if (account! = null) {
            account.withdraw(amount);
            transactionHistory[getAccountIndex(accountNumber)]. add (new Transaction
(transactionHistory.length + 1, accountNumber, amount, TransactionType.WITHDRAWAL));
        } else {
            System.out.println("Account not found.");
        }
    }
    public List<Transaction> getTransactionHistory(int accountNumber) {
        int index = getAccountIndex(accountNumber);
        if (index! = -1) {
            return transactionHistory[index];
        }
        return null;
    }

    private int getAccountIndex(int accountNumber) {
        for (int i = 0; i < numAccounts; i++) {
            if (accounts[i]. getAccountNumber () == accountNumber) {
                return i;
            }
        }
        return -1; // Account not found
    }
}
```

```
}
```

Account Class

```
import java.util.ArrayList;
import java.util.List;

public abstract class Account {
    protected int accountNumber;
    protected double balance;
    private List<Transaction> transactions;

    public Account (int accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0;
        this.transactions = new ArrayList<>();
    }

     public Account () {
        this (000);
     }

    public double getBalance() {
        return balance;
    }

    public int getAccountNumber() {
        return accountNumber;
    }
    public void recordTransaction(Transaction transaction) {
        transactions.add(transaction);
    }

    public List<Transaction> getTransactions() {
        return new ArrayList<>(transactions);
    }

    public abstract void deposit (double amount);
    public abstract void withdraw (double amount);
}
```

CheckingAccount Class

```
public class CheckingAccount extends Account {

    // Default Transaction Fee
    private static double FEE = 100.568;
    private int transactionId;

    // default constructor
    public CheckingAccount() {
        super ();
    }
```

```
/**
 * Parameter constructor to initialize CheckingAccount
 * with a custom Account Number and a Customer Transaction
 * Fee.
 */
public CheckingAccount(int accountNumber, double fee) {
   super(accountNumber);
   FEE = fee;
}
/**
 * Function to deposit funds into the account as long as the amount
 *  parameter is > 0
 * Apply Transaction fee for the CheckingAccount
 */
@Override
public void deposit (double amount) {
   // First Check amount
   if (amount > 0) {
      balance += amount;
      System.out.printf("Amount %.2f deposited%n", amount);

      // Apply Transaction Fee
      balance -= FEE;
      System.out.printf("Fee %.2f Applied%n", FEE);
      System.out.printf("Current Balance is: %.2f%n", balance);

      // Record deposit transaction
```

```
            recordTransaction(new Transaction(transactionId++, getAccountNumber(), amount,
TransactionType.DEPOSIT));
        } else {
            System.out.println("A negative amount cannot be deposited");
        }
    }

    /**
     * Function to withdraw funds from the Account as long as 1. Amount to withdraw
     * must be > 0 2. Amount to withdraw must be <= balance
     */
    @Override
    public void withdraw (double amount) {
        // Same check
        if (amount > 0) {
            // Check sufficient balance
            if ((amount + FEE) <= balance) {
                System.out.printf("Amount of %.2f withdrawn from Account%n", amount);
                balance -= amount;
                balance -= FEE;
                System.out.printf("Fee of %.2f applied%n", FEE);
                System.out.printf("Current Balance is: %.2f%n", balance);

                // Record withdrawal transaction
           recordTransaction(new Transaction(transactionId++, getAccountNumber(), amount,
TransactionType.WITHDRAWAL));
            }
        } else {
            System.out.println("Negative amount cannot be withdrawn!");
        }
    }
}
```

SavingsAccount Class

```
public class SavingsAccount extends Account {

    // interest rate
    private double interestRate;
    private int transactionId;

    // default constructor
    public SavingsAccount() {
        super ();
    }

    /**
     * Parameter constructor to initialize Savings account with a customer account
     * number and interest rate
     */
    public SavingsAccount(int accountNumber, double interestRate) {
        super(accountNumber);
        this.interestRate = interestRate;
    }

    // getter function
    public double getInterestRate() {
        return this.interestRate;
    }

    public void setInterestRate(double interestRate) {
        this.interestRate = interestRate;
    }

    public double calcInterest() {
        return balance * interestRate;
    }

    public void applyInterest() {
        double interest = calcInterest();
        System.out.printf("Interest amount %.2f added to balance%n", interest);
        deposit(interest);

        // Record interest transaction
        recordTransaction(new Transaction(transactionId++, getAccountNumber(), interest,
TransactionType.INTEREST));
    }

    /**
     * Function to deposit funds into the account as long as the amount
     * parameter is > 0
     * Apply Transaction fee for the CheckingAccount
     */

    @Override
    public void deposit (double amount) {
        // First Check amount
        if (amount > 0) {
            balance += amount;
            System.out.printf("Amount %.2f deposited%n", amount);
            System.out.printf("Current Balance is: %.2f%n", balance);

            // Record deposit transaction
            recordTransaction(new Transaction(transactionId++, getAccountNumber(), amount,
TransactionType.DEPOSIT));
        } else {
            System.out.println("A negative amount cannot be deposited");
        }
    }

    /**
     * Function to withdraw funds from the Account as long as
     * 1. Amount to withdraw
     * must be > 0
     * 2. Amount to withdraw must be <= balance
     */
    @Override
    public void withdraw (double amount) {
        // Same check
        if (amount > 0) {
            // Check sufficient balance
            if (amount <= balance) {
                System.out.printf("Amount of %.2f withdrawn from Account%n", amount);
                balance -= amount;
                System.out.printf("Current Balance is: %.2f%n", balance);


                // Record withdrawal transaction
                recordTransaction(new Transaction(transactionId++, getAccountNumber(), amount,
TransactionType.WITHDRAWAL));
            }
        } else {
            System.out.println("Negative amount cannot be withdrawn!");
        }
    }
}
```

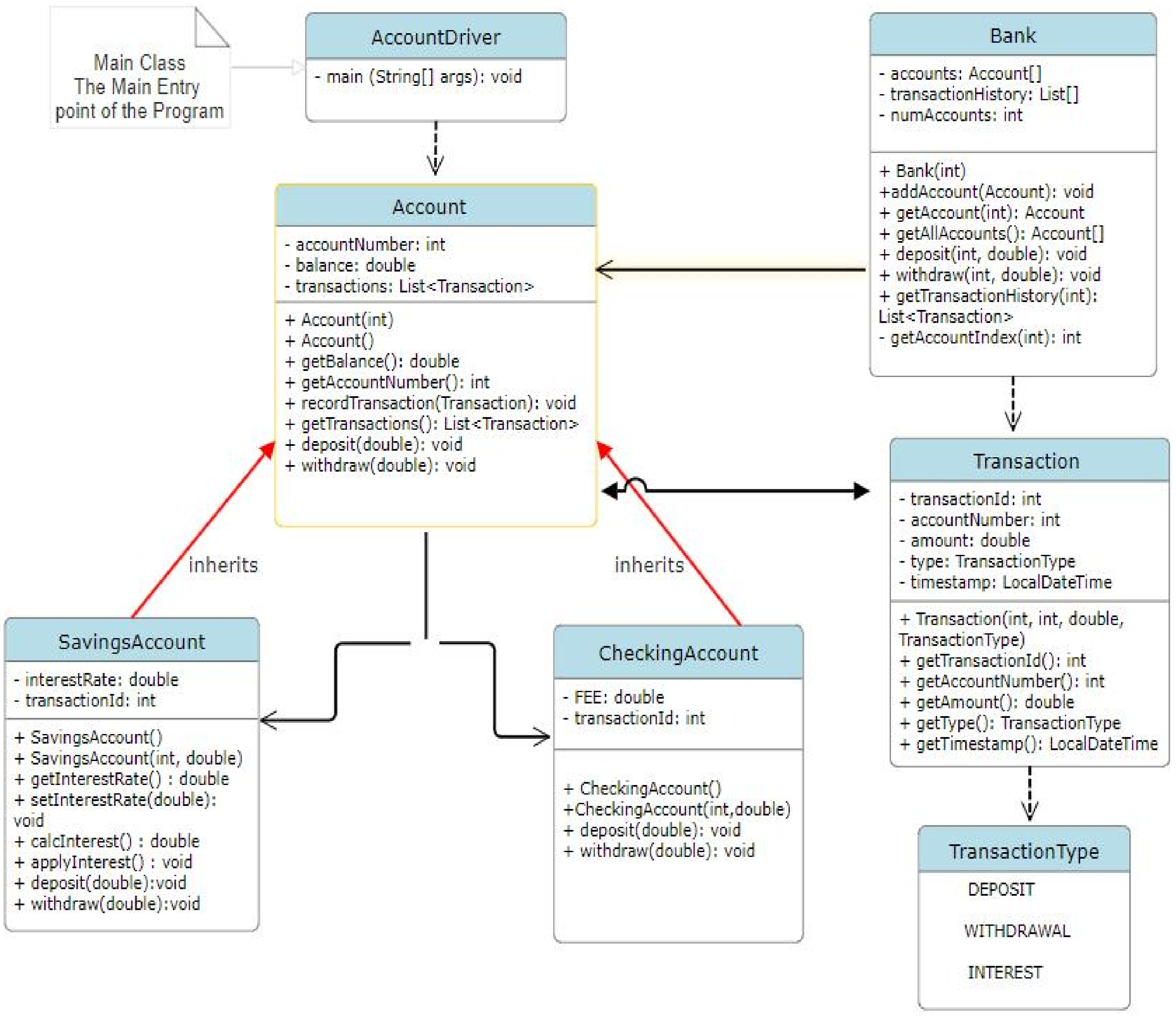
Transaction Class

```
import java.time.LocalDateTime;
import java.time.ZoneId;

public class Transaction {
    private int transactionId;
    private int accountNumber;
    private double amount;

    private TransactionType type;

    private LocalDateTime timestamp;


    public Transaction (int transactionId, int accountNumber, double amount, TransactionType
type) {

        this.transactionId = transactionId;

        this.accountNumber = accountNumber;

        this.amount = amount;

        this.type = type;

        this.timestamp = LocalDateTime.now(ZoneId.of("Asia/Dhaka")); // Set to Bangladesh time
zone

    }


    // Getters for transaction details

    public int getTransactionId() {

        return transactionId;

    }


    public int getAccountNumber() {

        return accountNumber;

    }


    public double getAmount() {

        return amount;

    }


    public TransactionType getType() {
        return type;
    }

    public LocalDateTime getTimestamp() {
        return timestamp;
    }

    @Override
    public String toString() {
        return "Transaction ID: " + transactionId +
                ", Type: " + type +
                ", Amount: $" + amount +
                ", Timestamp: " + timestamp;
    }
}
```

TransactionType Enum

```
public enum TransactionType {
    DEPOSIT,
    WITHDRAWAL, INTEREST
}
```

Diagram:


Main Class
The Main Entry
point of the Program
AccountDriver
- main (String[] args): void
Bank
- accounts: Account[]
- transactionHistory: List[]
- numAccounts: int
+ Bank(int)
+addAccount(Account): void
+ getAccount(int): Account
+ getAllAccounts(): Account[]
+ deposit(int, double): void
+ withdraw(int, double): void
+ getTransactionHistory(int): List<Transaction>
- getAccountIndex(int): int
Account
- accountNumber: int
- balance: double
- transactions: List<Transaction>
+ Account(int)
+ Account()
+ getBalance(): double
+ getAccountNumber(): int
+ recordTransaction(Transaction): void
+ getTransactions(): List<Transaction>
+ deposit(double): void
+ withdraw(double): void
inherits
inherits
Transaction
- transactionId: int
- accountNumber: int
- amount: double
- type: TransactionType
- timestamp: LocalDateTime
+ Transaction(int, int, double, TransactionType)
+ getTransactionId(): int
+ getAccountNumber(): int
+ getAmount(): double
+ getType(): TransactionType
+ getTimestamp(): LocalDateTime
SavingsAccount
- interestRate: double
- transactionId: int
+ SavingsAccount()
+ SavingsAccount(int, double)
+ getInterestRate() : double
+ setInterestRate(double): void
+ calcInterest() : double
+ applyInterest() : void
+ deposit(double):void
+ withdraw(double):void
CheckingAccount
- FEE: double
- transactionId: int
+ CheckingAccount()
+CheckingAccount(int,double)
+ deposit(double): void
+ withdraw(double): void
TransactionType
DEPOSIT
WITHDRAWAL
INTEREST

Explanation:

Code Structure and Organization:

Initially, structuring the code to accommodate various functionalities of the banking system posed a challenge. To address this, the code was organized into separate classes based on their responsibilities, such as Account, Transaction, Bank, CheckingAccount, SavingsAccount, and AccountDriver. This allowed for a clear separation of concerns and facilitated modular development.

Challenges Faced and Solutions:

During development, a key challenge was designing classes and establishing relationships accurately to represent real-world banking concepts while adhering to object-oriented principles. Deciding on class structures and defining relationships required careful analysis and iterative refinement. To address this, we employed a systematic approach to class design and relationship establishment, conducting thorough domain analysis to identify key entities and their interactions. We iteratively reviewed and refined class structures to accurately reflect real-world banking scenarios while maintaining adherence to object-oriented principles.

Object-Oriented Principles:

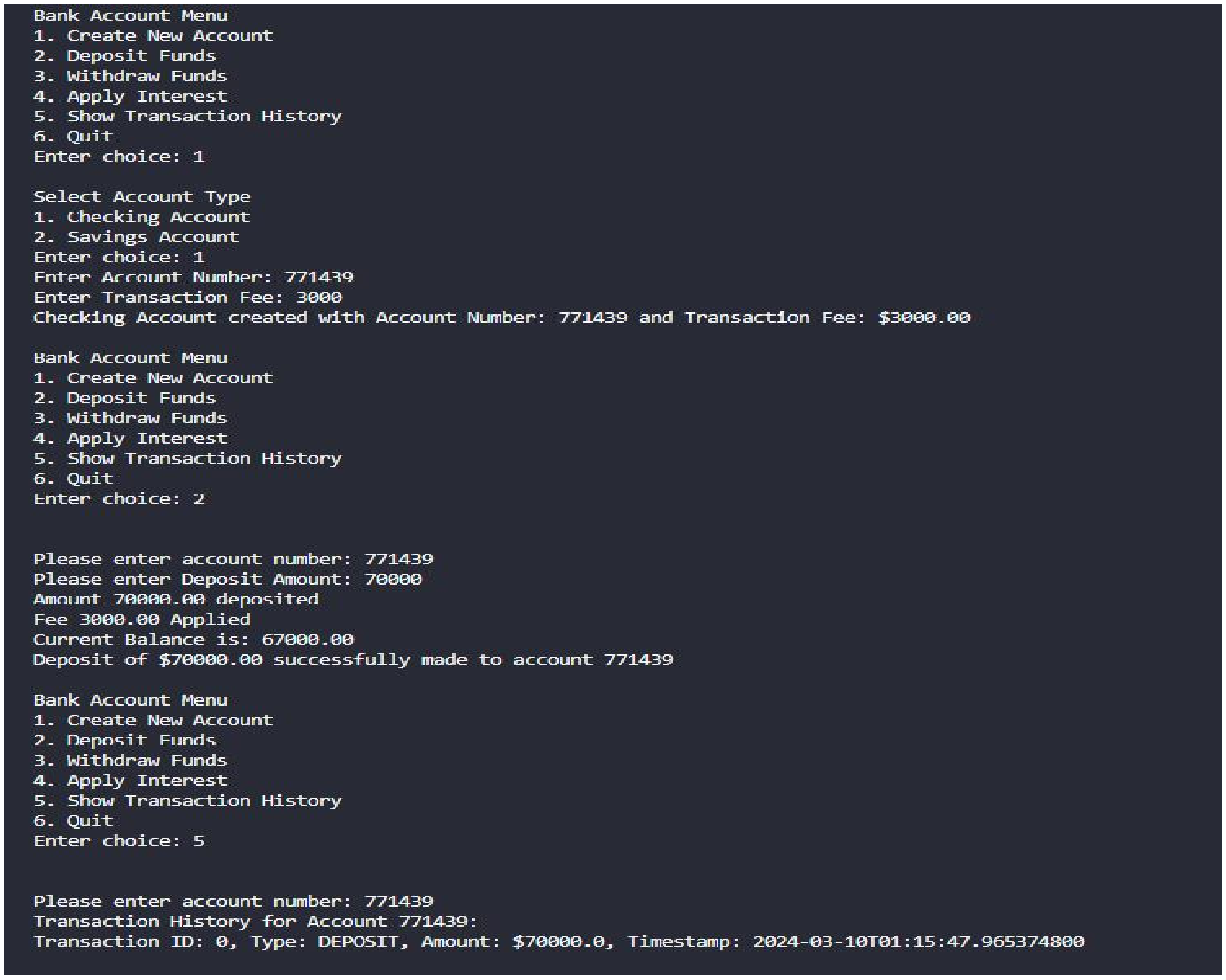
Inheritance: Inheritance was utilized extensively in the codebase to promote code reusability and accommodate diverse account types. For example, the CheckingAccount and SavingsAccount classes inherit from the Account class, allowing for the reuse of common attributes and behaviors while providing specific implementations for each account type.

Method Overloading: Method overloading was employed in the CheckingAccount and SavingsAccount classes to provide tailored implementations for specific account types. For instance, overloading the deposit and withdraw methods allowed for accommodating transaction fees and interest calculations, respectively.

Method Overriding: Method overriding was used in subclasses to provide specific implementations of inherited methods. For example, the applyInterest method was overridden in the SavingsAccount class to calculate and apply interest based on the account balance.

Polymorphism: Polymorphism played a crucial role in promoting flexibility and extensibility in the codebase. By defining common behaviors in the superclass and allowing subclasses to override these behaviors as needed, the system achieved adaptability to accommodate different account types and transaction operations.

Test:



```
Bank Account Menu
1. Create New Account
2. Deposit Funds
3. Withdraw Funds
4. Apply Interest
5. Show Transaction History
6. Quit
Enter choice: 1

Select Account Type
1. Checking Account
2. Savings Account
Enter choice: 1
Enter Account Number: 771439
Enter Transaction Fee: 3000
Checking Account created with Account Number: 771439 and Transaction Fee: $3000.00

Bank Account Menu
1. Create New Account
2. Deposit Funds
3. Withdraw Funds
4. Apply Interest
5. Show Transaction History
6. Quit
Enter choice: 2


Please enter account number: 771439
Please enter Deposit Amount: 70000
Amount 70000.00 deposited
Fee 3000.00 Applied
Current Balance is: 67000.00
Deposit of $70000.00 successfully made to account 771439

Bank Account Menu
1. Create New Account
2. Deposit Funds
3. Withdraw Funds
4. Apply Interest
5. Show Transaction History
6. Quit
Enter choice: 5


Please enter account number: 771439
Transaction History for Account 771439:
Transaction ID: 0, Type: DEPOSIT, Amount: $70000.0, Timestamp: 2024-03-10T01:15:47.965374800
```

This scenario tests the functionality of depositing funds into a checking account and applying transaction fees in the Java-based bank management application we developed.

Conclusion:

Engineering Knowledge: The development of this Java application for handling a bank system demands a solid understanding of object-oriented programming concepts and Java development, including inheritance, polymorphism, error handling, and input validation. It also requires familiarity with banking concepts and practices to ensure the accuracy and reliability of the system.

Technical Challenges: While the creation of this Java application does not face wide-ranging or conflicting technical issues, it does entail addressing challenges in class design, relationship establishment, and error handling mechanisms. Implementing robust input validation and error handling techniques was crucial to prevent unexpected behavior and maintain data integrity.

Abstract Thinking and Analysis: Formulating this Java application required abstract thinking and analysis to design a flexible and extensible banking management system. This involved identifying common behaviors and attributes across different account types, implementing inheritance and polymorphism to promote code reusability, and addressing various edge cases through meticulous error handling.

Learning Opportunities: Developing this Java application provided valuable learning opportunities in object-oriented design principles, Java programming, and real-world application development. It reinforced the importance of modularity, encapsulation, and abstraction in creating maintainable and scalable software systems. Additionally, it enhanced skills in problem-solving, error handling, and collaborative development, laying a strong foundation for future software engineering endeavors.

In summary, the development of this banking management system underscored the application of object-oriented principles and Java programming techniques. By addressing real-world banking requirements and implementing robust error handling mechanisms, the solution exemplifies our commitment to delivering reliable and scalable software systems. As we continue to refine and enhance our understanding of software engineering principles, this assignment serves as a testament to our commitment to delivering innovative and robust solutions to complex problems in the realm of financial management.